Incrementally fast updated frequent pattern trees

Tzung-Pei Hong a,*, Chun-Wei Lin b, Yu-Lung Wu b

a Department of Electrical Engineering, National University of Kaohsiung, Kaohsiung 811, Taiwan, ROC
b Department of Information Management, I-Shou University, Kaohsiung 84008, Taiwan, ROC

Abstract

The frequent-pattern-tree (FP-tree) is an efficient data structure for association-rule mining without generation of candidate itemsets. It was used to compress a database into a tree structure which stored only large items. It, however, needed to process all transactions in a batch way. In real-world applications, new transactions are usually inserted into databases. In this paper, we thus attempt to modify the FP-tree construction algorithm for efficiently handling new transactions. A fast updated FP-tree (FUFP-tree) structure is proposed, which makes the tree update process become easier. An incremental FUFP-tree maintenance algorithm is also proposed for reducing the execution time in reconstructing the tree when new transactions are inserted. Experimental results also show that the proposed FUFP-tree maintenance algorithm runs faster than the batch FP-tree construction algorithm for handling new transactions and generates nearly the same tree structure as the FP-tree algorithm. The proposed approach can thus achieve a good trade-off between execution time and tree complexity.

© 2007 Elsevier Ltd. All rights reserved.

Keywords: Data mining; FP-tree; FUFP-tree; Incremental mining; Maintenance

1. Introduction

Years of effort in data mining have produced a variety of efficient techniques (Agrawal, Imielenki, & Swami, 1993b; Chen, Han, & Yu, 1996). Depending on the type of databases processed, these mining approaches may be classified as working on transaction databases, temporal databases, relational databases, and multimedia databases, among others. On the other hand, depending on the classes of knowledge derived, the mining approaches may be classified as finding association rules, classification rules, clustering rules, and sequential patterns (Agrawal & Srikant, 1995), among others. Among them, finding association rules in transaction databases is most commonly seen in data mining (Agrawal et al., 1993a; Agrawal & Srikant, 1994; Agrawal, Srikant, & Vu, 1997; Fukuda, Morimoto, Morishita, & Tokuyama, 1996; Han & Fu, 1995; Mannila, Toivonen, & Verkamo, 1994; Park, Chen, & Yu, 1997; Srikant & Agrawal, 1995; Srikant & Agrawal, 1996).

In the past, many algorithms for mining association rules from transactions were proposed, most of which were based on the Apriori algorithm (Agrawal et al., 1993a), which generated and tested candidate itemsets level-by-level. This may cause iterative database scan and high computational cost.

Han et al. thus proposed the frequent-pattern-tree (FP-tree) structure for efficiently mining association rules without generation of candidate itemsets (Han, Pei, & Yin, 2000). The FP-tree (Han et al., 2000) was used to compress a database into a tree structure which stored only large items. It was condensed and complete for finding all the frequent patterns. The construction process was executed tuple by tuple, from the first transaction to the last one. After that, a recursive mining procedure called FP-Growth was executed to derive frequent patterns from
the FP-tree. They showed the approach could have a better performance than Apriori.

Both the Apriori and the FP-tree mining approaches belong to batch mining. That is, they must process all the transactions in a batch way. In real-world applications, new transactions are usually inserted into databases. In this case, the originally desired large itemsets may become invalid, or new large itemsets may appear in the resulting updated databases (Cheung, Han, Ng, & Wong, 1996; Cheung, Lee, & Kao, 1997; Lin & Lee, 1998; Sarda & Srinivas, 1998; Zhang, 1999). Designing an efficient algorithm that can maintain association rules as a database grows is thus critically important.

One noticeable incremental mining algorithm was the fast updated algorithm (called FUP), which was proposed by Cheung et al. (1996) for avoiding the shortcomings mentioned above. The FUP algorithm modified the Apriori mining algorithm (Agrawal & Srikant, 1994) and adopted the pruning techniques used in the DHP (Direct Hashing and Pruning) algorithm (Park et al., 1997). It first calculated large itemsets mainly from newly inserted transactions, and compared them with the previous large itemsets from the original database. According to the comparison results, FUP determined whether re-scanning the original database was needed, thus saving some time in maintaining the association rules.

As mentioned above, the process for generating large itemsets from the FP-tree is much faster than the Apriori algorithm. When new transactions come, the FP-tree mining algorithm must re-process the entire updated databases to form the correct FP-tree. In this paper, we thus attempt to modify the batch procedure of the FP-tree algorithm for incremental mining based on the FUP concept. A fast updated FP-tree (FUFP-tree) structure is proposed, which will make the tree update easier. It is similar to the FP-tree structure except that the links between parent nodes and their child nodes are bi-directional. Besides, the counts of the sorted frequent items are also kept in the Header_Table of the FP-tree algorithm. Bi-directional linking and storing the counts in the Header_Table will help fasten the maintenance process. An incremental FUFP-tree maintenance algorithm is then proposed for processing newly inserted transactions. It first partitions items into four parts according to whether they are large or small in the original database and in the new transactions. Each part is then processed in its own way. The Header_Table and the FUFP-tree are correspondingly updated whenever necessary. Experimental results also show the proposed FUFP-tree maintenance algorithm has a good performance.

The remainder of this paper is organized as follows. Related works are reviewed in Section 2. The proposed FUFP structure and maintenance algorithm are described in Section 3. An example is given to illustrate the proposed algorithm in Section 4. Experimental results for showing the performance of the proposed algorithm are provided in Section 5. Conclusions are given in Section 6.

2. Review of related works

In this section, some related researches about mining of association rules are briefly reviewed. They are data mining process, the FUP algorithm, and the FP-tree algorithm.

2.1. The data mining process for association rules

Data mining involves applying specific algorithms to extract patterns or rules from data sets in a particular representation. One common type of data mining is to derive association rules from transaction data, such that presence of certain items in a transaction will imply the presence of some other items. To achieve this purpose, Agrawal and his co-workers proposed several mining algorithms based on the concept of large itemsets to find association rules from transaction data (Agrawal et al., 1993a; Agrawal & Srikant, 1994; Agrawal et al., 1997). They divided the mining process into two main phases. In the first phase, candidate itemsets were generated and counted by scanning the transaction data. If the count of an itemset appearing in the transactions was larger than the pre-defined threshold value (called the minimum support), the itemset was considered a large itemset. Itemsets containing only one item were processed first. Large itemsets containing only single items were then combined to form candidate itemsets containing two items. This process was repeated until all large itemsets had been found. In the second phase, association rules were induced from the large itemsets found in the first phase. All possible association combinations for each large itemset were formed, and those with calculated confidence values larger than a predefined threshold (called the minimum confidence) were output as association rules.

2.2. The FUP algorithm

In real-world applications, transaction databases usually grow over time and the association rules mined from them must be re-evaluated. Some new association rules may be generated and some old ones may become invalid. Conventional batch-mining algorithms solve this problem by re-processing the entire new databases when new transactions are inserted into original databases. They, however, require lots of computational time and waste existing mined knowledge.

Cheung et al. thus proposed the FUP algorithm (Cheung et al., 1996) to effectively handle new transactions for maintaining association rules. Considering an original database and some new inserted transactions, the following four cases (illustrated in Fig. 1) may arise:

Case 1: An itemset is frequent both in an original database and in newly inserted transactions.
Case 2: An itemset is frequent in an original database but not frequent in newly inserted transactions.
Case 3: An itemset is not frequent in an original database but frequent in newly inserted transactions.
Case 4: An itemset is not frequent both in an original database and in newly inserted transactions.

Since itemsets in Case 1 are large in both the original database and the new transactions, they will still be large after the weighted average of the counts. Similarly, itemsets in Case 4 will still be small after the new transactions are inserted. Thus Cases 1 and 4 will not affect the final large itemsets. Case 2 may remove existing large itemsets, and Case 3 may add new large itemsets.

The FUP algorithm thus maintains large itemsets by considering the above four cases. In FUP, large itemsets with their counts in preceding runs are recorded for later use in maintenance. As new transactions are added, FUP first scans them to generate candidate 1-itemsets (only for these transactions), and then compares these itemsets with the previous ones. FUP partitions candidate 1-itemsets into two parts according to whether they are large for the original database. If a candidate 1-itemset from the newly inserted transactions is also among the large 1-itemsets from the original database, its new total count for the entire updated database can easily be calculated from its current count and previous count since all previous large itemsets with their counts are kept by FUP. Whether an original large itemset is still large after new transactions are inserted is determined from its updated support. By contrast, if a candidate 1-itemset from the newly inserted transactions does not exist among the large 1-itemsets in the original database, one of two possibilities arises. If this candidate 1-itemset is not large for the new transactions, then it cannot be large for the entire updated database, which means no action is necessary. If this candidate 1-itemset is large for the new transactions but not among the original large 1-itemsets, the original database must be re-scanned to determine whether the itemset is actually large for the entire updated database. Using the processing tactics mentioned above, FUP is thus able to find all large 1-itemsets for the entire updated database. After that, candidate 2-itemsets from the newly inserted transactions are formed and the same procedure is used to find all large 2-itemsets. This procedure is repeated until all large itemsets have been found. A summary of the four cases and their FUP results is given in Table 1.

FUP is thus able to handle Cases 1, 2 and 4 more efficiently than conventional batch mining algorithms.

### 2.3. The frequent pattern tree

Han et al. proposed the FP-tree for efficiently mining association rules without generation of candidate itemsets (Han et al., 2000). The FP-tree mining algorithm consists of two phases. The first phase focuses on constructing the FP-tree from the database, and the second phase focuses on deriving frequent patterns from the FP-tree. They are described below.

#### 2.3.1. Construction of an FP-tree

The FP-tree (Han et al., 2000) is used to compress a database into a tree structure storing only large items. It is condensed and complete for finding all the frequent patterns. Three steps are involved in FP-tree construction. The database is first scanned to find all items with their frequency. The items with their supports larger than a pre-defined minimum support are selected as large 1-itemsets (items). Next, the large items are sorted in descending frequency. At last, the database is scanned again to construct the FP-tree according to the sorted order of large items. The construction process is executed tuple by tuple, from the first transaction to the last one. After all transactions are processed, the FP-tree is completely constructed.

The Header_Table is also built to help tree traversal. The Header_Table includes the sorted large items and their pointers (called frequency head) linking to their first occurrence nodes in the FP-tree. If more than one node have the same item name, they are also linked in sequence. Note that the links between nodes are single-directional from parents to children.

Below, a simple example is given to illustrate the process of the FP-tree construction. Assume there are five transactions shown in Table 2. Also assume the minimum support is set at 50%. The FP-tree is constructed in the following way.

First, the database is scanned to find large items. All the items with their counts are shown in Table 3, in which the large items are marked.

From Table 3, it can be observed that the set of large 1-itemsets, named $L_1$, includes \{a:3, b:3, c:4, f:4, m:3, p:3\}, where the number after an item represents its count. Next, the items in $L_1$ are sorted according to their frequency.
descending frequency. The sorted $L_1$, named $L_0$ is \{f: 4, c: 4, a: 3, b: 3, m: 3, p: 3\}. At last, the database is scanned again to construct the FP-tree. The transactions with only sorted large items are shown in Table 4 for illustrating the construction process easily.

In Table 4, the first transaction is (f, c, a, m, p). The root of the FP-tree is first set Null. This transaction is then inserted into the FP-tree as the first branch. Each node in the branch is attached a count of 1. The results after the first transaction is processed are shown in Fig. 2.

The second transaction is next processed. It shares the same prefix (f, c, a) as the first branch of the FP-tree. The counts of nodes f, c and a are then incremented by 1, and a new node (b:1) is created and linked to (a:2) as its child. Another new node (m:1) is then created and linked to (b:1). Besides, a link is created between the two nodes of m. The results after the second transaction is processed are shown in Fig. 3.

The same process is then executed for the other transactions. After all the transactions are processed, the resulting Header_Table and FP-tree are shown in Fig. 4.

2.3.2. Mining of large itemsets

After the FP-tree is constructed from a database, a mining procedure called FP-Growth (Han et al., 2000) is executed to find all large itemsets. FP-Growth does not need to generate candidate itemsets for mining, but derives frequent patterns directly from the FP-tree. It is a recursive process, handling the frequent items one by one and bottom-up according to the Header_Table. A conditional FP-tree is generated for each frequent item, and from the tree the large itemsets with the processed item can be recursively derived.

Specifically, a conditional FP-tree is generated in the following way. Let a prefix path of an item $I$ in the FP-tree be the preceding part of a branch above $I$. The corresponding prefix paths for a large item $I$ are first extracted from the FP-tree. The count of each node in a prefix path is set as the count of $I$ in the same branch. The counts of an item appearing in different prefix paths are then summed up. The items with their counts larger than or equal to the

<table>
<thead>
<tr>
<th>TID</th>
<th>Sorted frequent items</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>f, c, a, m, p</td>
</tr>
<tr>
<td>200</td>
<td>f, c, a, b, m</td>
</tr>
<tr>
<td>300</td>
<td>f, b</td>
</tr>
<tr>
<td>400</td>
<td>c, b, p</td>
</tr>
<tr>
<td>500</td>
<td>f, c, a, m, p</td>
</tr>
</tbody>
</table>

Table 2
A database with five transactions

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>a, c, d, f, g, i, m, p</td>
</tr>
<tr>
<td>200</td>
<td>a, b, c, f, l, m, o</td>
</tr>
<tr>
<td>300</td>
<td>b, f, h, j, o</td>
</tr>
<tr>
<td>400</td>
<td>b, c, k, s, p</td>
</tr>
<tr>
<td>500</td>
<td>a, c, e, f, l, m, n, p</td>
</tr>
</tbody>
</table>

Table 3
All the items with their counts

<table>
<thead>
<tr>
<th>Item</th>
<th>Frequency</th>
<th>Item</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
<td>j</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>k</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>l</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>m</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
<td>o</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td>p</td>
<td>3</td>
</tr>
<tr>
<td>h</td>
<td>1</td>
<td>s</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>w</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 4. The resulting Header_Table and FP-tree in the example.

Fig. 2. The FP-tree after the first transaction is processed.

Fig. 3. The FP-tree after the second transaction is processed.
minimum count are then selected to build the conditional FP-tree for \( I \). Each prefix path, like a transaction, is used to build the conditional FP-tree as in the FP-tree construction. A conditional FP-tree is thus a little like a sub-FP-tree with the processed item lying at its leaves. An itemset composed of the original item \( I \) and each item in the conditional FP-tree is thus certainly large. The process is recursively executed until all the items in a conditional FP-tree are processed.

Below, the FP-tree formed in the previous example is used to illustrate the FP-Growth procedure. The frequent items in the Header_Table in Fig. 4 are processed bottom-up and one by one. Item \( p \) is first processed. Two prefix paths exist for item \( p \): \((f:4)(c:4)(a:3)(m:2)(p:2)\) and \((c:1)(b:1)(p:1)\). The counts of all the nodes in the first prefix path are then updated as 2 since they appear only twice with item \( p \) in the branch. Similarly, the counts of the nodes in the second prefix path are all updated as 1 since they appear only once with item \( p \). Thus, two converted prefix paths are \((f:2)(c:2)(a:2)(m:2)\) and \((c:1)(b:1)\). The counts of the items in the two prefix paths are then calculated. After that, only item \( c \) (with count 3) is large. The conditional FP-tree for item \( p \) is shown in Fig. 5. The frequent patterns with \( p \) are \((p:3)\) and \((cp:3)\).

Next, item \( m \) is processed. There are two converted prefix paths for item \( m \): \((f:2)(c:2)(a:2)\) and \((f:1)(c:1)(a:1)(b:1)\). Only the counts of items \( f \), \( c \), and \( a \) are larger than or equal to the minimum count (3). The set of large items for the conditional FP-tree of item \( m \) are thus \( \{f, c, a\} \). The conditional FP-tree for item \( m \) is then generated by inserting the prefix paths with only the large items one by one. The results are shown in Fig. 6.

The frequent patterns with \( m \) are \((m:3)\), \((am:3)\), \((cm:3)\) and \((fm:3)\). A conditional FP-tree is then recursively constructed in the sequence of \( am \), \( cm \), and \( fm \). The prefix path for \( am \) is \((f:3)(c:3)\). The conditional FP-tree for \( am \) is shown in Fig. 7. The large itemsets with \( am \) are \((am:3)\), \((cam:3)\) and \((fam:3)\).

The same process is then recursively executed to find all the conditional FP-trees, and to derive the large itemsets from the trees. After all the items on the conditional FP-tree of \( m \) have been processed, the following frequent patterns with \( m \) are derived: \((m:3)\), \((fm:3)\), \((cm:3)\), \((fm:3)\), \((fam:3)\), \((cam:3)\) and \((fcam:3)\). All the other frequent items in the Header_Table can be processed in the same way.

Many mining methods for finding association rules based on the FP-tree structure have also been proposed. Qiu et al. proposed the QFP-growth mining approach to mine association rules (Qiu, Lan, & Xie, 2004). Mohammad proposed the COFI-tree structure to replace the conditional FP-tree (Zaiane & Mohammed, 2003). Ezeife constructed a generalized FP-tree, which stored all the large and non-large items, for incremental mining without rescanning databases (Ezeife, 2002). Some related researches are still in progress.

3. The proposed incremental FUFP-tree maintenance approach

An FUFP-tree must be built in advance from the original database before new transactions come. The FUFP-tree construction algorithm is the same as the FP-tree algorithm (Han et al., 2000) except that the links between parent nodes and their child nodes are bi-directional. Bi-directional linking will help fasten the process of item deletion in the maintenance process. Besides, the counts of the sorted frequent items are also kept in the Header_Table.

When new transactions are added, the proposed incremental maintenance algorithm will process them to maintain the FUFP-tree. It first partitions items into four parts according to whether they are large or small in the original database and in the new transactions. Each part is then processed in its own way. The Header_Table and
the FUFP-tree are correspondingly updated whenever necessary.

In the process for updating the FUFP-tree, item deletion is done before item insertion. When an originally large item becomes small, it is directly removed from the FUFP-tree and its parent and child nodes are then linked together. On the contrary, when an originally small item becomes large due to the new transactions, its updated support is usually only a little larger than the minimum support. The FUFP-tree can thus be least updated in this way, and the performance of the proposed maintenance algorithm can be greatly improved. The entire FUFP-tree can be re-constructed in a batch way when a sufficiently large number of transactions are inserted. The notation used in this paper is first described below.

3.1. Notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>the original database</td>
</tr>
<tr>
<td>$T$</td>
<td>the set of new transactions</td>
</tr>
<tr>
<td>$U$</td>
<td>the entire updated database, i.e., $D \cup T$</td>
</tr>
<tr>
<td>$t$</td>
<td>the number of transactions in $T$</td>
</tr>
<tr>
<td>$I$</td>
<td>an item</td>
</tr>
<tr>
<td>$S^d(I)$</td>
<td>the number of occurrences of $I$ in $D$</td>
</tr>
<tr>
<td>$S^T(I)$</td>
<td>the number of occurrences of $I$ in $T$</td>
</tr>
<tr>
<td>$S^U(I)$</td>
<td>the number of occurrences of $I$ in $U$</td>
</tr>
<tr>
<td>$sup$</td>
<td>the support threshold for large itemsets</td>
</tr>
<tr>
<td>$Insert_Items$</td>
<td>the set of items with which the new transactions are reprocessed for updating the FUFP-tree</td>
</tr>
<tr>
<td>$Rescan_Items$</td>
<td>the set of items with which the original transactions are reprocessed for updating the FUFP-tree</td>
</tr>
<tr>
<td>$Rescan_Transactions$</td>
<td>the set of original transactions with items in the $Rescan_Items$</td>
</tr>
</tbody>
</table>

The details of the proposed algorithm are described below.

3.2. The proposed algorithm

**INPUT:** An old database, its corresponding Header Table storing the frequent items in descending order, its corresponding FUFP-tree, a support threshold $sup$ and a set of $t$ new transactions.

**OUTPUT:** A new FUFP-tree for the updated database.

**STEP 1:** Scan the new transactions to get all the items and their counts.

**STEP 2:** Check whether the items are large in the new transactions by comparing their counts in the new transactions with the minimum count ($t*sup$).

**STEP 3:** For each item $I$ which are large both in the new transactions and in the original database (appearing in the Header Table), do the following substeps (Case 1):

Substep 3-1: Set the new count $S^U(I)$ of $I$ in the entire updated database as:

$$S^U(I) = S^d(I) + S^T(I),$$

where $S^d(I)$ is the count of $I$ in the Header Table (original database) and $S^T(I)$ is the count of $I$ in the new transactions.

Substep 3-2: Update the count of $I$ in the Header Table as $S^U(I)$.

Substep 3-3: Put $I$ in the set of $Insert\_Items$, which will be further processed in STEP 6.

**STEP 4:** For each item $I$ which are small in the new transactions but large in the original database (appearing in the Header Table), do the following substeps (Case 2):

Substep 4-1: Set the new count $S^U(I)$ of $I$ in the entire updated database as:

$$S^U(I) = S^d(I) + S^T(I).$$

Substep 4-2: If $S^d(I) \geq (d + t)*sup$, item $I$ will still be large after the database is updated; update the count of $I$ in the Header Table as $S^U(I)$ and add $I$ to the set of $Insert\_Items$.

Substep 4-3: If $S^d(I) < (d + t)*sup$, item $I$ will become small after the database is updated; Remove $I$ from the Header Table, connect each parent node of $I$ directly to the corresponding child node of $I$, and remove $I$ from the FUFP-tree.

**STEP 5:** For each item $I$ which are large in the new transactions but small in the original database (not appearing in the Header Table), do the following substeps (Case 3):

Substep 5-1: Rescan the original database to find out the transactions with item $I$, and calculate the count $S^d(I)$ of $I$ in the original database.

Substep 5-2: Set the new count $S^U(I)$ of $I$ in the entire updated database as:

$$S^U(I) = S^d(I) + S^T(I),$$

where $S^d(I)$ is the count of $I$ obtained from Substep 5-1 and $S^T(I)$ is the count of $I$ in the new transactions.

Substep 5-3: If $S^U(I) \geq (d + t)*sup$, item $I$ will be large after the database is updated; Add item $I$ both in the set of $Insert\_Items$ and in the set of $Rescan\_Items$, and put the transaction IDs with item $I$ in the set of $Rescan\_Transactions$.

**STEP 6:** Sort the items in the $Rescan\_Items$ in a descending order of their updated counts.

**STEP 7:** Insert the items in the $Rescan\_Items$ to the end of the Header Table according to the descending order of their counts.
STEP 8: For each original transaction in the Rescan_Transactions with an item $J$ existing in the Rescan_Items, if $J$ has not been at the corresponding branch of the FUFP-tree for the transaction, insert $J$ at the end of the branch and set its count as 1; Otherwise, add 1 to the count of the node $J$.

STEP 9: For each new transaction with an item $J$ existing in the Insert_Items, if $J$ has not been at the corresponding branch of the FUFP-tree for the new transactions, insert $J$ at the end of the branch and set its count 1; Otherwise, add 1 to the count of $J$ node.

In Step 8, a corresponding branch is the branch generated from the large items in a transaction and corresponding to the order of items appearing in the Header_Table. After Step 9, the final updated FUFP-tree is constructed. The new transactions can then be integrated into the original database. Based on the FUFP-tree, the desired association rules can then be found by the FP-Growth mining approach as proposed in Han et al. (2000).

4. An example

In this session, an example is given to illustrate the proposed incremental mining approach for maintaining an FUFP tree when new transactions are inserted. Table 5 shows a database to be used in the example. The database contains 10 transactions and 9 items, denoted $a$ to $i$.

Assume the support threshold is set at 50%. For the given database, the large 1-itemsets are $a$, $b$, $f$, $g$ and $h$, from which the Header_Table can be constructed. The FUFP-tree are then formed from the database and the Header_Table, with the results shown in Fig. 8.

Assume the five new transactions shown in Table 6 appear. The proposed incremental mining algorithm proceeds as follows.

STEP 1: The five new transactions are first scanned to get the items and their counts. The results are shown in Table 7.

STEP 2: The counts of the items appearing in the new transactions are checked against with the minimum count. In this case, the minimum support is set at 0.5. The minimum count for an item to be large in the new transactions is thus $5 \times 0.5 = 2.5$. In this example, items $a$, $b$, $c$ and $d$ are large in the new transactions, and the remaining ones are small.

STEP 3: The items which are large both in the new transactions and in the original database are processed. In this example, items $a$ and $b$ satisfy the condition and are processed. Take item $a$ as an example to illustrate the substeps. The count of item $a$ in the Header_Table is 8, and the count in the new transactions is 3. The new count of item $a$ is thus $8 + 3 = 11$. The frequency value of item $a$ in the Header_Table is changed as 11. Item $a$ is then put into the set Insert_Items. After STEP 3, Insert_Items = \{a, b\}.

STEP 4: The items which are small in the new transactions but large in the original database are processed. In this example, items $f$, $g$ and $h$ satisfy the condition...
and are processed. The minimum count for an item to be large in the updated database is 7.5. Take item $g$ first as an example to illustrate the substeps. The count of item $g$ in the Header_Table is 6, and the count in the new transactions is 2. The new count of item $g$ is thus $6 + 2 = 8$, larger than the minimum count. $g$ is thus still large for the updated database, and is put into the set of $\text{Insert} \_ \text{Items}$. The frequency value of item $g$ in the Header_Table is changed as 8. Moreover, both the new counts of items $f$ and $h$ in the Header_Table are calculated as 7, smaller than the minimum count. Items $f$ and $h$ are thus removed from the Header_Table. In this case, the FUFP-tree needs to be processed as well. Take $h$ as an example to illustrate the substeps. The Header_Table and the FUFP-tree before $h$ is processed are shown in Fig. 9, with all the nodes for $h$ being marked. The results after $h$ is processed are shown in Fig. 10.

**STEP 5:** The items which are large in the new transactions but small in the original database (not appearing in the Header_Table) are processed. In this example, items $c$ and $d$ satisfy the condition and will be processed. The original database is then rescanned to find the transactions with items $c$ and $d$ and their counts. The counts of items $a$, $b$, and $c$ are, respectively, 3 and 4 in the original database.

The updated count of item $c$ after the database is updated is thus $3 + 4 = 7$, smaller than the minimum count. $c$ is thus a small item and will not affect the Header_Table and the FUFP-tree. It is thus directly ignored. On the contrary, the count of item $d$ is $4 + 4 = 8$, larger than the minimum count. $d$ is thus a large itemset after the database is updated. Item $d$ is then inserted into the Header_Table. $d$ is then put into both the sets of $\text{Insert} \_ \text{Items}$ and $\text{Rescan} \_ \text{Items}$. After **STEP 5**, $\text{Insert} \_ \text{Items} = \{a, b, g\}$, and $\text{Rescan} \_ \text{Items} = \{d\}$, and $\text{Rescan} \_ \text{Transactions} = \{1, 3, 6, 8\}$. The corresponding transactions with their IDs in the $\text{Rescan} \_ \text{Transactions}$ are shown in Table 8.

**STEP 6:** The items in the set of $\text{Rescan} \_ \text{Items}$ are sorted in descending order of their updated counts. In this example, $\text{Rescan} \_ \text{Items}$ contains only $d$, and no sorting is needed.

**STEP 7:** The items in the $\text{Rescan} \_ \text{Items}$ are inserted into the end of the Header_Table. In this example, $d$ is thus inserted. The Header_Table after this step is shown in Fig. 12.

**STEP 8:** The FUFP-tree is updated according to the original transactions with items existing in the $\text{Rescan} \_ \text{Items}$. In this example, $\text{Rescan} \_ \text{Items} = \{d\}$. The corresponding branches for the original transactions with $d$ are shown in Table 9.

The first branch is then processed. This branch shares the same prefix $(b, a, g)$ as the current FUFP-tree. A new node $(d:1)$ is thus created and linked to $(g:3)$ as its child. The results after

---

Table 8

<table>
<thead>
<tr>
<th>Transaction No.</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a, b, c, d, g, h, e$</td>
</tr>
<tr>
<td>3</td>
<td>$b, d, e, f, g$</td>
</tr>
<tr>
<td>6</td>
<td>$a, c, d, e, g, h$</td>
</tr>
<tr>
<td>8</td>
<td>$b, c, d, f, g$</td>
</tr>
</tbody>
</table>
the first branch is processed are shown in Fig. 13. Note that the counts for items b, a, and g are not increased since they have already been counted in the construction of the previous FUFP-tree. The same process is then executed for the other three corresponding branches. The final results are shown in Fig. 14.

STEP 9: The FUFP-tree is updated according to the new transactions with items existing in the Insert_Items. In this example, Insert_Items = \{b, a, g, d\}. The corresponding branches for the new transactions with any of these items are shown in Table 10.

<table>
<thead>
<tr>
<th>Transaction no.</th>
<th>Items</th>
<th>Corresponding branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, b, c, d, e, g, h</td>
<td>b, a, g, d</td>
</tr>
<tr>
<td>2</td>
<td>b, d, e, f, g</td>
<td>b, g, d</td>
</tr>
<tr>
<td>3</td>
<td>a, c, d, e, g, h</td>
<td>a, g, d</td>
</tr>
<tr>
<td>4</td>
<td>b, c, d, e, g, h</td>
<td>b, g, d</td>
</tr>
</tbody>
</table>

The first branch shares the same prefix (b, a, g, d) as the current FUFP-tree. The counts for items b, a, g, and d are then increased by 1 since they have not yet counted in the construction of the previous FUFP-tree. The results after the first branch is processed are shown in Fig. 15.

The second branch shares the same prefix (b, a) as the current FUFP-tree. A new node (d:1) is thus created and linked to (a:9) as its child. The results after the second branch is processed are shown in Fig. 16.
The same process is then executed for the other two branches. The final results are shown in Fig. 17. Based on the FUFP-tree shown in Fig. 17, the desired large itemsets can then be found by the FP-Growth mining approach as proposed in Han et al. (2000). Item \(d\) is first processed. The following five prefix paths exist for item \(d\): 

- \((b:13) (a:5)\)
- \((b:13) (a:10)\)
- \((b:13) (g:6)\)
- \((b:13)\) and \((a:1)(g:1)\)

The counts of all the nodes in the first prefix path are then updated as 3 since they appear three times with item \(d\) in the branch. Similarly, the count of the nodes in the other prefix paths is updated as 1, 2, 1 and 1, respectively. Thus, the five converted prefix paths are 

- \((b:3)(a:3)(g:3)\)
- \((b:12)(a:1)(g:1)\)
- \((b:12)(g:2)\)
- \((b:12)\) and \((a:1)(g:1)\).

The counts of the items in the five prefix paths are then calculated. In this example, no large itemsets with \(d\) are generated because their counts are all smaller than the minimum count (which is 8). All the other frequent items in the Header_Table are then processed in the same way. The results are shown in Table 11.

From the results in Table 11, there is only a large 2-itemset \(ab\) generated from the database.

### 5. Experimental results

Experiments were made to compare the performance of the batch FP-tree construction algorithm and the incremental FUFP-tree maintenance algorithm for processing new transactions. When new transactions came, the batch FP-tree construction algorithm integrated new transactions into the original database and constructed a new FP-tree from the updated database. The process was executed whenever new transactions came. The incremental FUFP-tree maintenance algorithm processed new transactions incrementally in the way mentioned in Section 3.

The experiments were performed in C++ on an Intel x86 PC with a 2.8 GHz processor and 512 MB main memory and running the Microsoft Windows XP operating system. A real dataset called **BMS-POS** (Zheng, Kohavi, & Mason, 2001) were used in the experiments. This dataset was also used in the KDDCUP 2000 competition. The **BMS-POS** dataset contained several years of point-of-sale data from a large electronics retailer. Each transaction in this dataset consisted of all the product categories purchased by a customer at one time. There were 515,597 transactions with 1657 items in the dataset. The maximal length of a transaction was 164 and the average length of the transactions was 6.5.

The first 400,000 transactions were extracted from the BMS-POS database to construct an initial FP-tree. The next 5000 transactions were then sequentially used each time as new transactions for the experiments. The minimum support was set at 4%. The execution times and the numbers of nodes obtained from both the batch FP-tree construction algorithm and the incremental FUFP-tree maintenance algorithm were compared. Fig. 18 shows the execution times required by the batch FP-tree construction algorithm and by the FUFP-tree maintenance algorithm for processing each 5000 new transactions.

The first 400,000 transactions were extracted from the BMS-POS database to construct an initial FP-tree. The next 5000 transactions were then sequentially used each time as new transactions for the experiments. The minimum support was set at 4%. The execution times and the numbers of nodes obtained from both the batch FP-tree construction algorithm and the incremental FUFP-tree maintenance algorithm were compared. Fig. 18 shows the execution times required by the batch FP-tree construction algorithm and by the FUFP-tree maintenance algorithm for processing each 5000 new transactions.

### Table 11

<table>
<thead>
<tr>
<th>Items</th>
<th>Prefix paths</th>
<th>Large itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>{(b:3)(a:3)(g:3), (b:12)(a:1)(g:1)}</td>
<td>φ</td>
</tr>
<tr>
<td>g</td>
<td>{(b:4, a:4), (b:3), (a:1)}</td>
<td>φ</td>
</tr>
<tr>
<td>a</td>
<td>{(b:10)}</td>
<td>{(b:10)}</td>
</tr>
<tr>
<td>b</td>
<td>φ</td>
<td>φ</td>
</tr>
</tbody>
</table>

The same process is then executed for the other two branches. The final results are shown in Fig. 17.

Based on the FUFP-tree shown in Fig. 17, the desired large itemsets can then be found by the FP-Growth mining approach as proposed in Han et al. (2000). Item \(d\) is first processed. The following five prefix paths exist for item \(d\): 

- \((b:13)(a:10)(g:5)\)
- \((b:13)(a:10), (b:13)(g:2), (b:13)\) and \((a:1)(g:1)\)

The counts of all the nodes in the first prefix path are then updated as 3 since they appear three times with item \(d\) in the branch. Similarly, the count of the nodes in the other prefix paths is updated as 1, 2, 1 and 1, respectively. Thus, the five converted prefix paths are 

- \((b:3)(a:3)(g:3)\)
- \((b:12)(a:1)(g:1)\)
- \((b:12)(g:2)\)
- \((b:12)\) and \((a:1)(g:1)\).

The counts of the items in the five prefix paths are then calculated. In this example, no large itemsets with \(d\) are generated because their counts are all smaller than the minimum count (which is 8). All the other frequent items in the Header_Table are then processed in the same way. The results are shown in Table 11.

From the results in Table 11, there is only a large 2-itemset \(ab\) generated from the database.
In Fig. 18, it easily observed that the execution time by the proposed approach was much less than that by the batch FP-tree construction algorithm for handling new transactions. Especially, when the transaction numbers in the original database became larger, the FUFP-tree maintenance algorithm had a better speed-up.

The FUFP-tree maintenance algorithm may generate a less concise tree than the FP-tree construction algorithm since the latter completely follows the sorted frequent items to build the tree. As mentioned above, when an originally small item becomes large due to new transactions, its updated support is usually only a little larger than the minimum support. It is thus reasonable to put a new large item at the end of the Header_Table. The difference between the FP and the FUFP tree-structures will thus not be significant. For showing this effect, the numbers of nodes between the two algorithms are shown in Fig. 19.

It is observed from Fig. 19 that the FUFP-tree maintenance algorithm generated nearly the same nodes as the FP-tree construction algorithm. The effectiveness of the FUFP-tree maintenance algorithm is thus acceptable.

6. Conclusion

In this paper, we have proposed the FUFP maintenance structure and algorithm to efficiently and effectively handle new transaction insertion in data mining. The FUFP-tree structure is the same as the FP-tree structure (Han et al., 2000) except that the links between parent nodes and their child nodes are bi-directional. Besides, the counts of the sorted frequent items are also kept in the Header_Table. These modifications will make the tree update process easier.

When new transactions are added, the proposed incremental maintenance algorithm processes them to maintain the FUFP-tree. It first partitions items into four parts according to whether they are large or small in the original database and in the new transactions. Each part is then processed in its own way. The Header_Table and the FUFP-tree are correspondingly updated whenever necessary. It is reasonable to insert a new large item at the end of the Header_Table since when an originally small item becomes large due to new transactions, its updated support is usually only a little larger than the minimum support.

Experimental results also show that the proposed FUFP-tree maintenance algorithm runs faster than the batch FP-tree construction algorithm for handling new transactions and generates nearly the same tree structure as the FP-tree algorithm. The proposed approach can thus achieve a good trade-off between execution time and tree complexity.

The FP-Growth mining procedure was used for mining from the FP-tree in the past. It can also be borrowed for mining from FUFP-tree. Both the FP and the FUFP tree structures can easily allow the FP-Growth procedure to mine desired rules for only specified items. In this case, the maintenance of the tree structures is especially important. In the future, we will attempt to discuss other issues on incremental mining problems.

References


